CMPSC 465 Data Structures & Algorithms
Fall 2023 Antonio Blanca and Yana Safonova

HW 12

1. (20 pts.) **Maximum Sum Contiguous Subsequence.**

Subproblems: Define an array of subproblems $D(i)$ for $0 \leq i \leq n$. $D(i)$ will be the largest sum of a (possibly empty) contiguous subsequence ending exactly at position i.

Algorithm and Recursion: The algorithm will initialize $D(0) = 0$ and update the $D(i)$'s in ascending order according to the rule:

$$D(i) = \max\{0, D(i-1) + a_i\}$$

The largest sum is then given by the maximum element $D(i)^*$ in the array D. The contiguous subsequence of maximum sum will terminate at i^*. Its beginning will be at the first index $j \leq i^*$ such that $D(j-1) = 0$, as this implies that extending the sequence before j will only decrease its sum.

Correctness: The contiguous subsequence of largest sum ending at i will either be empty or contain a_i. In the first case, the value of the sum will be 0. In the second case, it will be the sum of a_i and the best sum we can get ending at $i-1$, i.e. $D(i-1) + a_i$. Because we are looking for the largest sum, $D(i)$ will be the maximum of these two possibilities.

Running Time: The running time for this algorithm is $O(n)$, as we have n subproblems and the solution of each can be computed in constant time. Moreover, the identification of the optimal subsequence only requires a single $O(n)$ time pass through the array D.

2. (20 pts.) **Maximum Average Value Path.** One interesting observation is we need N-1 down moves and N-1 right moves to reach the destination (bottom right). So any path from the top left corner to the bottom right corner requires 2N – 2 cells. In the average value ratio, the denominator is fixed and we need to maximize the numerator. Therefore we need to find the maximum sum path. If dp[i][j] represents the maximum sum till cell (i, j) from (0, 0) then at each cell (i, j), we update dp[i][j] as below:

(i) For j = 0, the only possible way to reach this cell is by moving down. Thus,
dp[i][0] = dp[i-1][0] + cost[i][0] for all i from 1 to n.

(ii) For i =0, the only possible way to reach this cell is by moving right. Thus,
dp[0][j] = dp[0][j-1] + cost[0][j] for all j from 1 to n.

(iii) For the inner cells,
dp[i][j] = max(dp[i-1][j], dp[i][j-1]) + cost[i][j]

Once we get the maximum sum of all paths, we divide this sum by (2N – 2) to get the maximum average. This algorithm performs a constant time operation for each cell of the N by N matrix, so it runs in $O(N^2)$ time. This algorithm is correct because there are only two ways to enter any interior cell. Recursively, the maximum value possible to reach some cell must be the maximum value of either its top neighbor or its left neighbor, plus its own value.

3. (20 pts.) **Coin Toss.**

Subproblems: Define $L(i, j)$ to be the probability of obtaining exactly j heads amongst the first i coin tosses.

Algorithm and Recursion: By the definition of L and the independence of the tosses:

$$L(i,j) = p_i L(i-1,j-1) + (1-p_i)L(i-1,j) \quad j = 0,1,\ldots,i$$

We can then compute all $L(i,j)$ by first initializing $L(0,0) = 1$, $L(i,j) = 0$ for all $j < 0$, and $L(i,j) = 0$ for all $i = 0, j \geq 1$. Then we proceed incrementally (in the order $i = 1,2,\ldots,n$, with inner loop $j = 0,1,\ldots,i$). The final answer is given by $L(n,k)$.

Correctness and Running Time: The recursion is correct as we can get j heads in i coin tosses either by obtaining $j-1$ heads in the first $i-1$ coin tosses and throwing a head on the last coin, which takes place with probability $p_i L(i-1,j-1)$, or by having already j heads after $i-1$ tosses and throwing a tail last, which has probability $(1-p_i)L(i-1,j)$. Besides, these two events are disjoint, so the sum of their probabilities equals $L(i,j)$. Finally, computing each subproblem takes constant time, so the algorithm runs in $O(nk)$ time.

4. (20 pts.) **Longest Palindromic Subsequence.** *Subproblems:* Define variables $L(i,j)$ for all $1 \leq i \leq j \leq n$ so that, in the course of the algorithm, each $L(i,j)$ is assigned the length of the longest palindromic subsequence of string $a[i,\cdots,j]$.

Algorithm and Recursion: The recursion will then be:

$$L(i,j) = \max\left\{L(i+1,j), L(i,j-1), L(i+1,j-1) + \text{equal}(a_i,a_j)\right\}$$

where equal(a,b) is 2 if a and b are the same character and is 0 otherwise, The initialization is the following:

$$\begin{aligned} \forall i, 1 \leq i \leq n \ , &\qquad L(i,i) = 1 \\ \forall i, 1 \leq i \leq n-1 \ , &\quad L(i,i+1) = 2 \text{ if } a_i = a_{i+1}, \text{ else } 1 \end{aligned}$$

$L(1,n)$ is the length of the longest palindromic subsequence.

Correctness and Running Time: Consider the longest palindromic subsequence s of $a[i,\cdots,j]$ and focus on the elements a_i and a_j. There are then three possible cases:

- If both a_i and a_j are in s then they must be equal and $L(i,j) = L(i+1,j-1) + \text{equal}(a_i,a_j)$
- If a_i is not a part of s, then $L(i,j) = L(i+1,j)$.
- If a_j is not a part of s, then $L(i,j) = L(i,j-1)$.

Hence, the recursion handles all possible cases correctly. The running time of this algorithm is $O(n^2)$, as there are $O(n^2)$ subproblems and each takes $O(1)$ time to evaluate according to our recursion.

5. (20 pts.) **Semi-Global Sequence Alignment.**

(a) We want S_1 to be able to start anywhere, without penalty. This can be translated to $E(0,j) = 0$ for all $i = 0,\ldots n$, or the top row of the algorithm's table being initialized to zero instead of counting up from zero to n. This allows the algorithm to begin placing characters from S_1 (moving down the rows of the table) aligned with any character of S_2 (in any column) with no penalty. Because it is only a change to the initialization values of the edit distance algorithm, the running time and correctness of the recurrence relation are unaffected.

(b) Similar to (a), but we instead want the leftmost column to be initialized to zero. This allows us to start placing letters from S_2 in any row without penalty.

(c) We want S_1 to be able to end in the middle of S_2, instead of both strings needing to end together. The algorithm knows that S_1 has ended when the final row has been reached, which can happen in any column thanks to the new rule. So, instead of recovering our final answer from the bottom-rightmost cell, we look for the minimum value from any cell of the bottom row. Taking the minimum ensures that no penalties after reaching the end of S_1 are counted. This change only affects how the final answer is retrieved, changing that step from $O(1)$ to $O(m)$. However, this is dominated by the algorithm's $O(nm)$ running time. The correctness of the rest of the algorithm is unchanged.

(d) Similar to (c), but we instead iterate over all the cells in the rightmost column. This will give us the minimum edit distance counting only until S_2 has ended and takes $O(n)$ time. The rest of the answer is identical to (c).

Rubric:

Problem 1, 20 pts

- 4 pts: subproblem definition
- 4 pts: base case
- 8 pts: right recursion
- 2 pts: correctness proof
- 2 pts: running time analysis

Problem 2, 20 pts

- 4 pts: subproblem definition
- 4 pts: base case (left column, top row)
- 8 pts: right recursion
- 2 pts: correctness proof
- 2 pts: running time analysis

Problem 3, 20 pts

- 4 pts: subproblem definition
- 4 pts: base case
- 8 pts: right recursion
- 2 pts: correctness proof
- 2 pts: running time analysis

Problem 4, 20 pts

- 4 pts: subproblem definition
- 4 pts: correct base case (2 points for $L(i,i)$ case and $L(i,i+1)$ case, respectively)
- 8 pts: correctly identify all possible cases for the subproblem (3 points for each possible case of $L(i,j)$: $L(i+1,j)$, $L(i,j-1)$, and $L(i+1,j-1)+\text{equal}(a_i,a_j)$)
- 3 pts: explains the correctness of each of the 3 possible cases (1 point each)
- 1 pts: running time analysis

Problem 5, 20 pts All parts 5pts. 3 for the modification, 2 for a reasonable explanation.